可視化プログラミングの基礎(2) : 粒子ベースボリュームレンダリング

AUTHOR(S):

坂本, 尚久; 小山田, 耕二

CITATION:

坂本, 尚久 ...[et al]. 可視化プログラミングの基礎(2) : 粒子ベースボリュームレンダリング. 計算工学 2009, 14(1): 2008-2014

ISSUE DATE:

2009-01

URL:

http://hdl.handle.net/2433/153378

チュートリアル

最近では、情報爆発時代における不可欠な技術として情報可視化技術の開発や先進的利用が注目されています。今回は、高度な可視化技術として位置づけられる有限要素法向けボリュームレンダリングに関するチュートリアル（全4回シリーズ）の第2回目として、京都大学の坂本 尚久先生と小山田 耕二先生に解説していただきます。

可視化プログラミングの基礎（2）
粒子ベースボリュームレンダリング

坂本 尚久
小山田 耕二

1　はじめに

大規模分散並列環境で計算された有限要素法解析結果は、ひとつの計算ノードに集約することが困難であるので、分散環境に保存されたままでボリュームレンダリング計算を実施するための工夫が必要とされる。この工夫として、2つの戦略が考えられる。ひとつは、規則格子ボリューム上にサンプリングしてその結果を使って効率よくボリュームレンダリングを行う戦略である[1]。もうひとつは、分散して格納された部分ボリュームデータに対して既存のボリュームレンダリング法を適用し部分画像を作成し、それらを視点からの順序に基づき重畳処理する戦略である[2]。前者については、特に複雑な境界を持つモデルに対して、サンプリングに起因するアーチファクトが起こる可能性が大きい。後者については、部分ボリュームの境界に凹部分が含まれる可能性があるので、視点からの順序が定まらないことによるアーチファクトが起こる可能性が大きい。本チュートリアルでは、後者の問題点を解決するために開発された粒子ベースボリュームレンダリング（Particle-based Volume Rendering, PBVR）の説明を行う。

筆者紹介

さかもと なおひさ
平成13年（株）ケイ・ジー・ティー入社。平成19年京大大学院工学研究科博士課程了。平成20年京都大学特定助教。情報可視化に関する研究に従事。博士（工学）。電子情報通信学会、可視化情報学会各会員。

こやまだ こうじ
昭和60年京大大学院工学研究科了。同年日本IBM入社。平成10年岩手県立大学助教授、平成13年京都大学助教授、平成15年同教授。博士（工学）。情報可視化、設計最適化の研究に従事。IEEE CS、日本シミュレーション学会、情報処理学会各員。

2　準備

一般的に、ボリュームデータを対象とした可視化処理では、等値面の生成や流線の計算など、ボリュームデータから幾何データ（ジオメトリデータ）を作成し描画することが多い。ボリュームレンダリングでは、通常、幾何データを作成することなく直接描画を行うが、粒子ベースボリュームレンダリングでは、処理の対象となるボリュームデータから粒子群を生成し可視化を行う。可視化システムの構築においては、ボリュームデータおよびジオメトリデータを処理の対象として扱う（図1）。本章では、粒子ベースボリュームレンダリングを説明する準備として、本手法が取り扱うボリュームデータとジオメトリデータについて説明する。

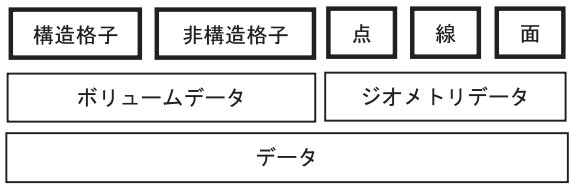


図1　可視化の対象となるデータ

2.1　ボリュームデータ

ボリュームデータは、前回のチュートリアル第2章でも説明したように、格子形状の違いにより構造格子型ボリュームデータと非構造格子型ボリュームデータに分類することができる。粒子ベースボリュームレンダリング（PBVR）では、構造格子および非構造格子ともに適用可能であるが、ここでは、より一般的な非構造格子型ボリュームデータを対象として説明する。

PBVRでは、隣接要素に関する情報を保持しておく必要がないため、ボリュームデータの属性として最低限以下のものを扱うことができれば処理を行うことができる。

・要素タイプ：要素の形状（四面体、六面体など）

・要素数：要素の数
・節点数：節点の数
・節点座標：節点の座標値配列
・数値データ：数値データ配列
・接続情報：要素を特定するための節点番号配列

要素を特定するための接続情報については、その要素を構成する節点番号の配列として表現され、各要素においては図2に示す順番に整列されている。ここで説明した非構造型ボリュームデータ構造は、次章で示すサンプルコード内で、UnstructuredVolumeObjectクラスとして表現している。

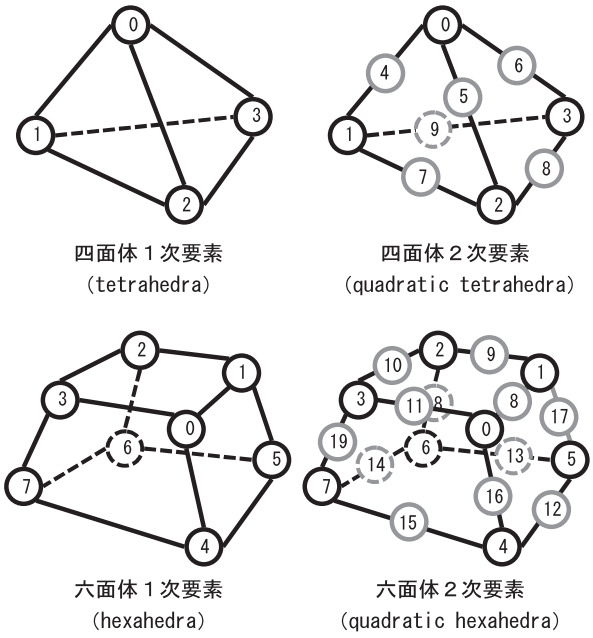


図2　要素内での節点の順番

2.2　ジオメトリデータ

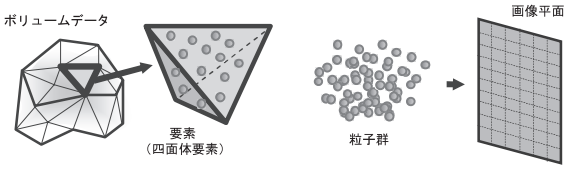
ジオメトリデータは、ディスプレイで表示するための基本的なデータ形状であり、一般的に、点データ・線データ・面データに分類される。ここでは、PBVRの処理対象となる点データについて説明する。PBVRを実行するために、最低限必要となる点データの属性は以下のとおりである。

・点数：点の数
・頂点座標：頂点の座標値配列
・色データ：頂点の色データ配列
・勾配ベクトル：節点の勾配ベクトル配列

線データについては、点データの属性に対して、線を構成するために必要な点の接続情報が必要であり、接続方法の違いによる線種に関する情報も必要となる。また、面データについても、点データの属性に対して、線データと同様に点の接続情報および面種(三角形や四角形)の情報が必要となる。ここで説明した点データ構造は、次章で示すサンプルコード内で、PointObjectクラスとして表現している。

3　粒子ベースボリュームレンダリング

粒子ベースボリュームレンダリングでは，前回導出した粒子密度(前回のチュートリアルの式(18)を参照)を満足するように不透明粒子を生成し，そのまま発光させるというSabellaのモデル[3]の原点に立ち帰ろうというものである。実装上としては、有限要素法プログラミングでなじみのある要素単位での積分計算を行い粒子を生成させ、単純に点を描画するだけのプログラムを書くことだけであり，極めて単純明快である。理論上の詳細については，既報論文[4]を参照されたい。



a. 要素ごとに粒子を生成　　b. 生成した粒子を画像平面に投影

図3　粒子ベースボリュームレンダリング

3.1　粒子生成処理

粒子生成処理は、粒子密度(詳細は前回のチュートリアルの式(18)を参照)を要素単位で積分することにより、要素毎に生成すべき粒子を決定し、以下のサンプルコードに示す粒子生成処理を行うことができる。

```
PointObject GenerateParticles(
   UnstructuredVolumeObject volume,
   TransferFunction tfunc )
{
   // 出力となる幾何形状データを構成する配列データ(頂点座標、色、法線)を
   // 準備する。
   std::vector<float> coords;
   std::vector<unsigned char> colors;
   std::vector<float> normals;

   // ボリュームデータに対して要素ごとの基本計算を行うためのクラス
   // を用意する。
   Cell cell( volume );

   // ボリュームデータを構成する要素数を取得する。
   int ncells = volume.ncells();

   // 要素ごとに粒子生成を行う。
   for ( int index = 0; index < ncells; ++index )
   {
      // index番目の要素をターゲットにする。
      cell.attachCell( index );

      // 要素の重心位置での粒子密度を計算する。
      float average_scalar = cell.averagedScalar();
      float density = CalculateDensity( average_degree, tfunc );

      // 生成粒子数を計算する。
      int nparticles = CalculateNumOfParticles( density, cell );

      // nparticle個の粒子を生成する。
      for ( int i = 0; i < nparticles; ++i )
      {
         // 要素内部に粒子を生成する(3次元位置(全体座標)を求める)。
         Vector3 coord = cell.randomSampling();

         // 生成粒子のスカラ値(補間値)を計算する。
         float scalar = cell.scalar();
         RGBColor color = tfunc.color( scalar );

         // 生成粒子の法線ベクトルを計算する。
         Vector3 normal( cell.gradient() );
```

```
        // 計算したデータを配列データに代入する。
        coords.push_back( coord.x() );
        coords.push_back( coord.y() );
        coords.push_back( coord.z() );

        colors.push_back( color.r() );
        colors.push_back( color.g() );
        colors.push_back( color.b() );

        normals.push_back( normal.x() );
        normals.push_back( normal.y() );
        normals.push_back( normal.z() );
      }
    }

    // 出力する点データを設定する。
    PointObject point;
    point.setCoords( coords );
    point.setColors( colors );
    point.setNormals( normals );

    return( point );
}
```

サンプルコード内のCellクラスは、要素単位での基本計算をサポートするクラスであり、その計算メソッドの具体的な実装については、多くの有限要素法解説書などに説明されており、また、前回のチュートリアルにおいてもその一部を解説しているため、ここでの説明は省略する。また、TransferFunctionクラスについては、指定されるスカラ値から色と不透明度を取得するためのメソッドが用意されている（前チュートリアルの3.3節を参照）。さらに、Vector3クラスおよびRGBColorクラスは、それぞれ、3次元ベクトル（x, y, z）に関する情報を保持しその計算をサポートするクラスと、色（red, blue, green）に関する情報を保持しその計算をサポートするクラスである。

(1) 粒子密度の計算

粒子密度の計算では、画素サイズを整数で除した大きさの直径dをもつ球状の粒子を仮定することにより、粒子発光モデルから粒子密度を推定する。この整数のことをサブピクセルレベル（*level*）と呼び、この値を大きく設定すると、粒子密度が大きくなり、その結果として画質が向上する。

$$d = \frac{1}{level} \tag{1}$$

また、粒子を表現するためにサブピクセルレベルに従って分割された*level* ^{2}個の部分領域をサブピクセルと呼ぶ。

粒子密度ρは、粒子の直径d、ユーザによって定められた伝達関数から計算される不透明度値α、そしてボリュームレイキャスティングで使用されるレイセグメント長さΔtを使って、以下のようにして計算することが可能である。

$$\rho = \frac{-\log(1-\alpha)}{d^2 \Delta t} \tag{2}$$

ただし、前回のチュートリアルの式（18）においては、粒子の半径をrとして、その投影面積をπr^2として計算していたが、ここでは、粒子の投影面積をサブピクセル面積d^2で近似して計算している。

ここで、空間点過程においてハードコア過程を考慮すると粒子密度に最大値ρ^{max}が存在し、その値は以下のようにして計算することができる。

$$\rho^{max} = \frac{1}{d^3} \tag{3}$$

したがって、不透明度にも対応する上限α^{max}が存在する。不透明度がα^{max}以上となる場合には対応する密度は一定値ρ^{max}となる。ここで$\alpha^{max}=1-\exp(-\pi r^2 \rho^{max} \Delta t)$である。以上のようにして、ユーザ指定の伝達関数によって導出される不透明度から粒子密度を計算することができる。

以上より、粒子密度の計算は、スカラ値と伝達関数を入力として以下のように定義される。

```
float CalculateDensity( float scalar, TransferFunction tfunc )
{
    // 大域的に定義されるパラメータを取得する。
    float subpixel_level = GetSubpixelLevel();
    float sampling_step = GetSamplingStep();

    // 式(1)の計算：
    // サブピクセルレベルから粒子の直径（サブピクセル長）を計算する。
    // ※CalculateSubpixelLength関数の実装は、利用するグラフィックスAPI
    // に依存するため、本チュートリアルでの詳細説明は省略する。
    float subpixel_length =
      CalculateSubpixelLength( subpixel_level );

    // 式(3)の計算：
    // 不透明度と粒子密度の最大値を計算する。
    float max_opacity =
      1.0 - exp( -sampling_step / subpixel_length );
    float max_density =
      1.0 / ( subpixel_length * subpixel_length * subpixel_length );

    // 伝達関数からスカラ値に対応する不透明度を計算する。
    float opacity = tfunc.opacity( scalar );

    // 式(2)の計算：
    // 計算された不透明度に対する粒子密度を計算する。
    float density = max_density;
    if ( opacity < max_opacity )
    {
      float tmp = subpixel_length*subpixel_length * sampling_step;
      density = - log( 1.0 - opacity ) / tmp;
    }

    return( density );
}
```

(2) 粒子数の計算

要素内部の生成粒子数Nは、粒子密度を要素単位で積分することによって、以下のようにして計算できる。

$$N = \int_{Cell} \rho \, dV \tag{4}$$

実際には生成粒子数は整数値であるため、最終的な生成粒子数nは、以下のようにして決定される。ただし、$[N]$は床関数を表し、実数Nに対してN以下の最大の整数を返す関数である。

$$n = \begin{cases} [N]+1 & \text{if } R \le N-[N] \\ [N] & \text{otherwise} \end{cases} \tag{5}$$

以上より、粒子数の計算は、推定した粒子密度と処理の対象とする要素を入力として、以下のようにして定義される。

```
float CalculateNumOfParticles( float density, Cell cell )
{
   // 要素の体積を計算する。
   float volume_of_cell = cell.volume();

   // 式(4)の計算：
   // 粒子数を計算する。
   float nparticles = density * volume_of_cell;
   float R = random();

   // 式(5)の計算：
   // 最終的な粒子数を決定する。
   int adjusted_nparticles = (int)nparticles;
   if ( nparticles - adjusted_nparticles < R )
   {
      ++adjusted_nparticles;
   }

   return( adjusted_nparticles );
}
```

3.2 粒子投影処理

生成した粒子は画像面に投影され、投影された粒子の座標はサブピクセル位置に離散化される(図4)。この離散化により、解像度W×Hの画像を生成するために、(W×*level*)×(H×*level*)の粒子格納領域が必要となる。粒子投影処理では、粒子は不透明であるために、投影順序は任意で良く、視点と粒子と距離(デプス)を基とした奥行き比較による陰点消去処理を行うだけでよい。粒子投影処理のサンプルコードを以下に示す。

```
void ProjectParticles( PointObject point )
{
    // 大域的に定義されるパラメータを取得する。
    float subpixel_level = GetSubpixelLevel();
    float W = GetImageWidth();
    float H = GetImageHeight();

    // 粒子数を取得する。
    int nvertices = point.nvertices();

    // 粒子群の頂点座標配列(配列の先頭ポインタ)を取得する。
    float* v  = point.coords().pointer();

    // 粒子格納領域(W*subpixel_level x H*subpixel_level)を用意する。
    Buffer buffer( W * subpixel_level, H * subpixel_level );

    // 粒子ごとに投影計算を行う。
    for( int index = 0; index < nvertices; index++ )
    {
        // index番目の粒子の座標を取得する。
        kvs::Vector3f M( v[index*3], v[index*3+1], v[index*3+2] );

        // 投影行列を計算する。
        // ※GetProjectionMatrix関数の実装は、利用するグラフィックスAPI
        // に依存するため、本チュートリアルでの詳細説明は省略する。
        kvs::Matrix44f P = GetProjectionMatrix();

        // 投影位置と奥行き値を計算する。
        // ※Project関数およびDepth関数の実装は、利用するグラフィックス
        // APIに依存するため、本チュートリアルでの詳細説明は省略する。
        kvs::Vector2f m = Project( P, M );
        float depth = Depth( P, M );

        // 粒子格納領域に投影粒子を格納する。このとき、既に投影される
        // サブピクセル位置に粒子が格納されていた場合、奥行き比較(デプス
        // テスト)を行い、視点位置により近いものを優先して格納する。
        if( buffer.depth( m ) > depth )
        {
            buffer.add( m );
        }
    }

    // サブピクセル処理を行う。
    SubpixelProcessing( buffer, subpixel_level );
}
```

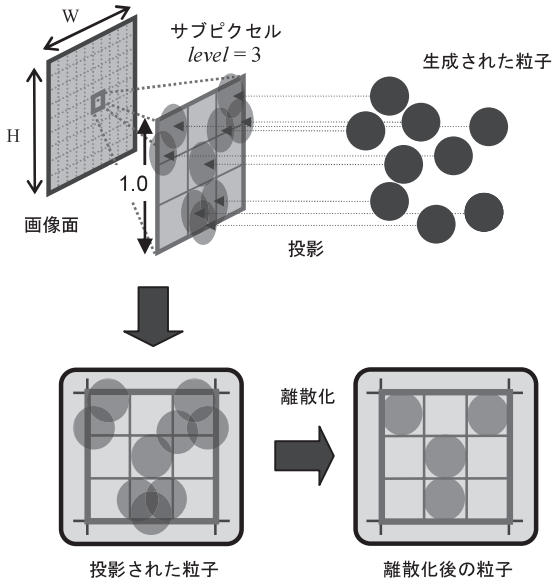


図4 投影粒子の離散化

サンプルコード内のBufferクラスは、粒子格納領域を表し、次節で説明するサブピクセル処理をサポートするクラスである。また、投影処理の計算においては、利用するグラフィックスAPIに依存する部分が多く、本チュートリアルでは、PBVRの基本的な振る舞いの理解を目的とするため、その詳細については省略した。本来、粒子(頂点)の投影計算やサブピクセル処理を含めた画素値計算(ラスタライズによる画素値の計算)については、グラフィックスAPIが標準的にまたは拡張として保有する機能(GPU(Graphics Processing Unit)を利用した処理)を利用することによって高速に計算することが可能である。このようなGPUの高速化機能を用いた実装については、次回のチュートリアルで詳しく説明する予定である。

(1) サブピクセル処理

先に述べたデプス値の比較による粒子隠蔽計算(陰点消去処理)は、サブピクセルごとに行われ、一つの画素を構成するサブピクセルでの投影粒子による輝度値を平均化することによってその画素値を決定することができる(図5)。

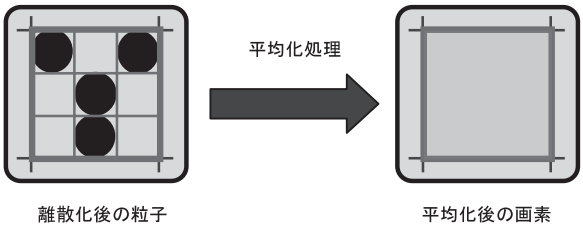


図5 サブピクセル処理

サブピクセル処理は、粒子格納領域とサブピクセルレベルを入力として以下のように定義される。

```
void SubpixelProcessing( Buffer buffer, float subpixel_level )
{
    // 大域的に定義されるパラメータを取得する。
    Camera camera = GetCamera();
    Light light = GetLight();
    float W = GetImageWidth();
    float H = GetImageHeight();

    // シェーディング計算クラスを準備する。
    Shader shader( camera, light );

    // 画素値を計算する。
    for( int  y = 0; y < H; y++ )
    {
        for( int x = 0; x < W; x++ )
        {
            float R = 0.0f;
            float G = 0.0f;
            float B = 0.0f;
            float A = 0.0f;
            float D = 0.0f;
            int npoints = 0;
            // サブピクセルの平均を計算する。
            for( int j = 0; j < subpixel_level; j++ )
            {
                for( int i = 0; i < subpixel_level; i++ )
                {
                    // 画素(x,y)のサブピクセル位置(i,j)に粒子が
                    // 格納されているかどうかを判定する。
                    if( buffer.hasParticle( x, y, i, j ) )
                    {
                        // 画素(x,y)のサブピクセル位置(i,j)に格納されて
                        // いる粒子の色、法線ベクトル、奥行き値を取得する。
                        RGBColor c = buffer.color( x, y, i, j );
                        Vector3 n = buffer.normal( x, y, i, j );
                        float d = buffer.depth( x, y, i, j );

                        // 輝度値の減衰率を計算しシェーディング処理を行う。
                        float a = shader.attenuation( n );
                        R += c.r() * a;
                        G += c.g() * a;
                        B += c.b() * a;

                        // Max関数は引数のうち大きいほうの値を返す。
                        D = Max( D, d );

                        npoints++;
                    }
                }
            }

            // 最終的な画素値とデプス値を計算する。
            float factor = 1.0 / ( subpixel_level * subpixel_level );
            R *= factor;
            G *= factor;
            B *= factor;
            A = npoints * factor;
            D = (npoints == 0) ? 1.0f : D;

            // 求めた画素値とデプス値を、対応する画素位置に書き込む。
            // ※SetPixel関数およびSetDepth関数は、利用するグラフィック
            // スAPIに依存するため、本チュートリアルでの詳細説明は省略する。
            SetPixel( x, y, R, G, B, A );
            SetDepth( x, y, D );
        }
    }
}
```

サンプルコード内のShaderクラスは、粒子の勾配ベクトルからカメラ(視点)およびライト(光源)の情報をもとにして計算される反射強度を用いてシェーディング処理を行うためのクラスである(前チュートリアルの3.3節を参照)。

4　可視化システムの実装

粒子ベースボリュームレンダリングを利用した非構造型ボリュームデータ向けの簡単な可視化システムを実装し、いくつかのテストデータに対する可視化結果を示す。本章では、可視化システムを構築する上で基本的な概念である可視化パイプラインについて説明した後、具体的なサンプルコードを示しながら、その実装について解説する。本章での可視化システムの実装には、C++の可視化基盤ライブラリKVS[5]を利用した。また、3章のサンプルコードで利用したいくつかのクラスについては、このKVS上に実装されている。クラスの対応を表1に示す。

表1　クラスの対応

サンプルコード内で利用されているクラス	KVS上で実装されているクラス
Cell	TetrahedralCell HexahedralCell QuadraticTetrahedralCell QuadraticHexahedralCell
Vector3	Vector3<T>
RGBColor	RGBColor
Buffer	ParticleBuffer
TransferFunction	TransferFunction
Shader	Shader::Phong

4.1　可視化パイプライン

可視化の目的は、実験や計測などから得られる抽象的なデータを視覚的に理解しやすい画像に変換することであり、この変換は複数の処理ステップに分割することができる[6]。これらの処理ステップは、データの流れに沿って順に実行され、それらをつなぎ合わせてできる一連の可視化処理手順は、データがまるでパイプを流れるかのごとく最終的に画像に変換されるため、可視化パイプラインと呼ばれている。一般的に、可視化パイプラインを構成する処理ステップは、フィルタリング、マッピング、レンダリングの3つの処理に分類される(図6)。

図6　可視化パイプライン

フィルタリング

データの削減など可視化対象となるデータの変換を行う。一般にジオメトリデータからジオメトリデータ、または、ボリュームデータからボリュームデータの変換を行う。

マッピング

ボリュームデータに対して色情報を割り当て、ジオメトリデータに変換する。色情報の割り当てには、伝達関数を利用する。

レンダリング

ボリュームデータまたはジオメトリデータから可視化結果画像を生成し、ディスプレイに表示する。

一般的に、可視化メソッドは上の3つの処理ステップのいずれかに分類され、それらを部品(モジュール)としてつなぎ合わせることによってさまざまな可視化処理を実現することが可能となる。

4.2 モジュールの実装

可視化パイプラインに基づき前節で説明した粒子ベースボリュームレンダリング(PBVR)をモジュールとして実装する。PBVRは、粒子生成処理と粒子投影処理の2つに分類することができた。

粒子生成処理は、ボリュームデータを入力として、ユーザ指定の伝達関数を基にして粒子密度を推定し、要素単位で粒子(ジオメトリデータ)を生成する処理であるため、マッピング処理に分類される。一方、粒子投影処理は、生成された粒子群を画像面に投影し、サブピクセル処理によって画素値を計算することで可視化結果画像を生成する処理であり、レンダリング処理に分類される。

(1) 粒子生成モジュール(マッピング)

KVSでは、粒子生成処理を行うマッピングモジュール(クラス)として、要素内に一様分布によって粒子を生成するクラス(kvs::CellByCellUniformSampling)と要素内の粒子密度分布に応じて粒子を生成するクラス(kvs::CellByCellMetropolisSampling)が実装されている。本チュートリアルの3.1節で説明した粒子生成処理を行うGenerateParticles関数の厳密な実装は、kvs::CellByCellUniformSamplingクラス内に定義されるgenerate_particlesメソッドに記述されている。

(2) 粒子投影モジュール(レンダリング)

KVSでは、粒子投影を行うレンダリングモジュール(クラス)として、kvs::ParticleVolumeRendererクラスを実装している。本チュートリアルの3.2節で説明した粒子投影処理を行うProjectParticles関数の厳密な実装については、kvs::ParticleVolumeRendererクラス内に定義されるproject_particlesメソッドに記述されている。

4.3 システムの構築

PBVRは、粒子生成処理を行うマッピングモジュールと粒子投影処理を行うレンダリングモジュールの2つのモジュールをつなぎ合わせることによって、可視化パイプラインを構築することができる(図7)。

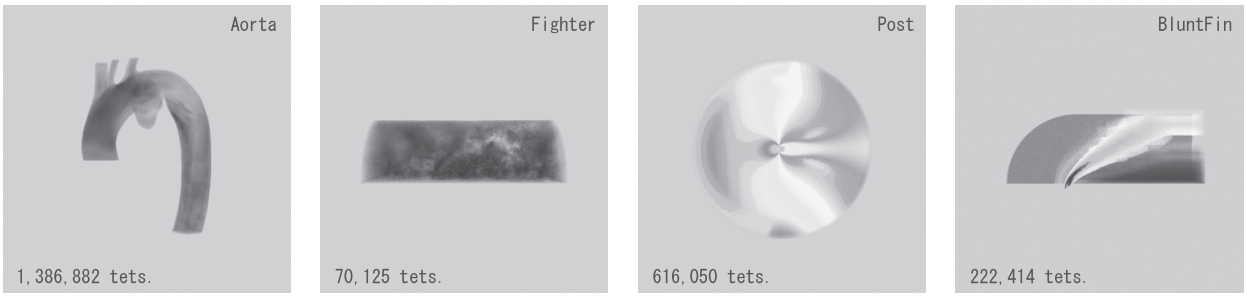
図7 粒子ベースボリュームレンダリング(可視化パイプライン)

KVSでは、可視化パイプラインの構築をサポートするためのクラス(kvs::VisualizationPipeline)を準備している。このクラスを利用して以下のようにして簡単に粒子ベースボリュームレンダリングシステムを開発することができる。

```
// 利用するクラスに対応するヘッダファイルをインクルートする。
#include <kvs/CellByCellUniformSampling>
#include <kvs/ParticleVolumeRenderer>
#include <kvs/VisualizationPipeline>
#include <kvs/glut/Global>
#include <kvs/glut/Screen>
#include <string>

int main( int argc, char** argv )
{
    // 大域変数クラスとスクリーンクラスを定義する。
    kvs::glut::Global global( argc, argv );
    kvs::glut::Screen screen( 512, 512 );

    // 引数で指定されるパラメータ(入力パラメータ)を取得する。
    //      第1引数:サブピクセルレベル
    //      第2引数:サンプリングステップ長
    //      第3引数:ボリュームデータファイル名
    //      第4引数:伝達関数ファイル名
    int subpixel_level = atoi( argv[1] );
    float sampling_step = atof( argv[2] );
    std::string data_filename = std::string( argv[3] );
    std::string tf_filename = std::string( argv[4] );

    // 伝達関数を準備する。
    kvs::TransferFunction tfunc( tf_filename );

    // 可視化パイプラインを準備する。同時に、読み込むファイル名を指定する。
    kvs::VisualizationPipeline pipe( data_filename );

    // 粒子生成モジュールを作成し、パラメータをセットする。
    kvs::PipelineModule m( new kvs::CellByCellUniformSampling );
    m.get<kvs::CellByCellUniformSampling>()
       ->setSubpixelLevel( subpixel_level );
    m.get<kvs::CellByCellUniformSampling>()
       ->setSamplingStep( sampling_step );
    m.get<kvs::CellByCellUniformSampling>()
       ->setTransferFunction( tfunc );

    // 粒子投影モジュールを作成し、パラメータをセットする。
      kvs::PipelineModule r( new kvs::ParticleVolumeRenderer );
    r.get<kvs::ParticleVolumeRenderer>()
       ->setSubpixelLevel( subpixel_level );

    // シェーディング処理クラスをセットする。
    float ka = 0.2f;
    float kd = 0.5f;
    float ks = 0.3f;
    float s = 20.0f;
    r.get<kvs::ParticleVolumeRenderer>()
        ->setShader( kvs::Shader::Phong( ka, kd, ks, s ) );

    // シェーディングを無効にするときは、以下のコメントアウトをはずす。
    // r.get<kvs::ParticleVolumeRenderer>()->disableShading();

    // モジュールをつなげて可視化パイプラインを構築する。
    pipe.connect( m ).connect( r );
    pipe.exec();

    // 可視化パイプラインをセットし、スクリーンを表示する。
    global.insert( pipe );
    screen.show();

    return( 0 );
}
```

4.4 実験

前節で述べた粒子ベースボリュームレンダリングシステムを利用して、いくつかのテストデータに対する可視化結果を図8に示す。この結果から、ボリュームデータ内部の数値データの空間的な変化が確認できる。図9には、サブピクセルレベルを変化させてレンダリングを行った結果を示す。この結果から、サブピクセルの増加に伴う画質向上が確認できる。また、サブ



(a) 大動脈瘤の影響による血流シミュレーション結果　(b) ジェット戦闘機まわりの気流シミュレーション結果　(c) 液化酸素（非圧縮性流体）の流体シミュレーション結果　(d) 鈍頭フィンの設置板上の気流シミュレーション結果

図8　PBVRシステムを用いた可視化結果

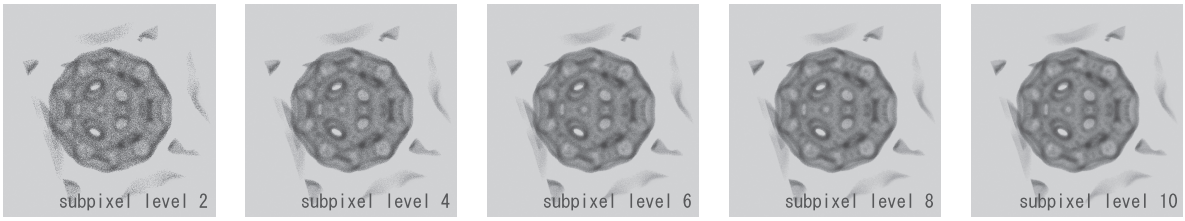


図9　サブピクセルレベルの変化にともなう画質の変化
（フラーレン分子の電子密度分布、要素タイプ：四面体、要素数：1,250,235）

ピクセルレベルを大きくすることによって生成粒子数が増加し、そのことによってレンダリング速度が低下する可能性があるが、動作環境に応じてサブピクセルレベルを調整することによって画像の詳細度を制御して可視化を行うことができる。

5　おわりに

本チュートリアルでは、要素単位での処理が可能であり、ユーザが指定する伝達関数から推定される粒子密度にしたがって生成される粒子群を用いて可視化を行う粒子ベースボリュームレンダリングを解説した。さらに、サンプルコードを示し具体的な実装方法についても説明し、簡単な粒子ベースボリュームレンダリングシステムの実装例を示した。

今回のサンプルコードを利用した説明においては、粒子ベースボリュームレンダリングの基本原理の理解をはかるために、粒子生成処理および粒子投影処理については、すべてCPU上で計算を行うことを想定した実装を行った。しかし、粒子投影処理については、GPUを利用した高速化技術を比較的容易に適用することができる。次回は、GPUを利用した粒子ベースボリュームレンダリングの高速化にフォーカスをあて解説する予定である。

■参考文献

[1] N. Sakamoto, K. Koyamada, K. Sakai, M. Kikugawa, Voxelization of Hexahedral Cell with the Two-Pass Rasterization Technique, The 4th IASTED International Conference on Visualization, Imaging, and Image Processing (VIIP2004), pp.178-181, (2004).

[2] Kwan-Liu Ma, Thomas W. Crockett, A scalable parallel cell-projection volume rendering algorithm for three-dimensional unstructured data, IEEE Parallel Rendering Symposium (1997).

[3] P. Sabella, A Rendering Algorithm for Visualizing 3D Scalar Fields, Computer Graphics, Vol.22, No.4, pp.51-58, (1988).

[4] 河村 拓馬、坂本 尚久、山崎 晃、小山田 耕二、粒子ベースボリュームレンダリングのための粒子密度推定法 －大規模非構造ボリュームデータに対する適用－、可視化情報学会論文集、Vol.28, No.11, pp.69-77, (2008).

[5] KVS: Kyoto Visualization System, http://www.viz.media.kyoto-u.ac.jp/kvs/

[6] R. B. Haber, and D. A. McNabb, Visualization idioms: A conceptual model for scientific visualization systems. In Visualization in Scientific Computing, pp.74-93, (1990).